Devoted entirely to ANALOG technology.
www.planetanalog.com

EmbeddedSystems
PROGRAMMING

Integer Square Roots

by Jack W. Crenshaw

Ask any math-smart programmer how to find a square root of a number, and five will get you ten, he'll tell you about Newton's method, which is an iterative approach. The general idea is this: given a number a for which we want to find the square root, we begin with an estimate x_0 for that root. We then refine the estimate using the formula:

$$x = \frac{1}{2}\left(x_0 + \frac{a}{x_0}\right) \quad (1)$$

This method does work, and works very well. With any kind of reasonable initial estimate, it converges very rapidly. But it is, after all, an iterative method, and the number of iterations it requires to converge depends on the accuracy of our initial guess. In the vicinity of the correct root, the method converges exponentially, meaning that it gives twice as many digits of accuracy at each step. However, if our initial guess is lousy, we only creep up on the solution by halving the error at each step. Everything depends on how well we make that initial guess. I wrote an article on this way back in 1991 ("Square Roots are Simple?" November 1991, p. 30)--my first column for *ESP*, in fact--where I discussed the intimate details of this method and how to optimize the guessing process.

Even with the best estimates, however, we can't predict in advance how many iterations the method will take to converge to suitable accuracy. In general, functions that execute an unpredictable number of iterations are frowned upon in real-time systems, for the obvious reason that they make the total time per task unpredictable, and introduce jitter, not to mention potential disaster in the form of task overruns, into the system.

If you're using integer arithmetic, there's yet another problem: the method fails to converge! Try it for $a = 15$ and you'll find that x oscillates between three and four. Only one of these numbers (the three) can possibly be correct, but tell that to Dr. Newton.

I suppose we could fix the method up easily enough, by refusing to allow a solution whose square is greater than a, but there's another possibility: a closed-form solution that can't oscillate. You and I both know that such algorithms exist--we all learned one in high school. And unless you're very weird, you promptly forgot it just as I did. As a sort of

dessert to our recent smorgasbord of algorithms for integer arithmetic, this month I'm offering a discussion of the closed-form solutions for the square root.

Begin at the Beginning

When I'm starting on a new algorithm, I sometimes find it's both fun and instructional to begin by trying to see just how simple-minded an algorithm I can write. I think I do this just to prove to myself that the problem can at least be solved. From there, anything else is mere refinement.

For the case of integer square roots, let's be specific about what we want: we want the *largest* integer *x* whose square is *less* than *a*. Well, I certainly know one way to find such an integer: an exhaustive search. We'll start with x = 1, keep incrementing it until it's too big, then back off by one. Listing 1 shows the simplest square root finder you will ever see. Note carefully the way the loop test is written. The "<=" sign is used, rather than simply "<", to guarantee that we always exit the loop with x too large.

```
LISTING 1
The simplest square root.

unsigned long sqrt(unsigned long a){
  unsigned long x = 1;
  while(x*x <= a)
    ++x;
  return <x;
}
```

Getting Better

Can we improve on Listing 1? Of course; it would be difficult *not* to do so. The biggest problem with the method is that it requires the multiplication *x*x* at each trial. This shouldn't be necessary. Since we're progressing systematically up through the list of all possible integers, we should be able to predict what the next square will be without having to actually perform the multiplication. We can do this by noting that:

$(n+1)^2 = n^2 + 2n + 1$ (2)

If we already have the square of *n*, which we do, we can get the square of the next larger number by adding $2n+1$. Now, does this expression look familiar? It should; it's the expression for the sequence of odd numbers. We can see this relationship graphically by considering the difference between the first few squares, as illustrated in Table 1.

```
Table 1 -- Squares and differences

                n                   n2       Difference
                0                   0
```

1	1	1
2	4	3
3	9	5
4	16	7
5	25	9
6	36	11
7	49	13
8	64	15
9	81	17
10	100	19

Equation 2 and Table 1 show us that we can generate each square simply by adding a successively larger odd integer to the previous one. Listing 2 shows the improved algorithm.

```
LISTING 2
Improving efficiency.

unsigned long sqrt(unsigned long a){
  unsigned long square = 1;
  unsigned long delta = 3;
  while(square <= a){
    square += delta;
    delta += 2;
  }
    return (delta/2 - 1);
}
```

This program may seem trivial, but the fact is, you could do a lot worse. While this approach, cycling through all possible integers, isn't going to be fast, it is simple and requires neither multiplication nor division. This makes it a wonderful candidate for implementation on a small microcontroller which doesn't support multiplication or division, where the word length is short, and speed isn't much of an issue. PIC programmers take note.

The Friden Algorithm

The next algorithm requires some background, which is best given in the form of a little ancient history. Once upon a time, a very long time ago, people didn't have ready access to computers. We didn't even have electronic calculators. I was working at NASA in those days. When we needed more accuracy than we could squeeze out of our slide rules, we used an electro-mechanical calculator. There were several vendors of such machines, but the most common for scientific work was a massive, motor-driven, electro-mechanical monster made by Friden. This thing was bigger than the largest typewriter, had a moveable carriage like one, cost about $1,500 in 1960 dollars, and had hernia-inducing mass. You didn't move a Friden without warming up first on a set of dumbbells. (I did once see, nevertheless, a Friden installed in the passenger side of an MG, for use in sports car rallies.)

The Friden keyboard was huge, consisting of 16 columns of 10 keys each, representing the 10 digits. You punched in a number by picking one from column *A*, one from column *B*, and so on. On the carriage was a set of rotating wheels, which displayed the answers as digits which showed up in little windows. A set of operator keys on the right told the computer what to do, and it did so with a great gnashing of gear teeth.

When there was a Friden in use in the room, everyone there knew it. The room lights dimmed, the motor whirred, the entire desktop shook, and the wheels spun with a sound somewhere between a threshing machine and a car in dire need of Mr. Transmission. It sounded as though the machine couldn't possibly last out the day, but in fact, the Friden was quite a reliable machine. The basic Friden was strictly a four-function calculator, with no memory except the little wheels on the carriage. However, a square root version was available at considerable extra cost. Our office had only one square root Friden, so we all had to take turns with it when we needed square roots. Or so I thought.

One day I was busily thinking deep-space thoughts, while my officemate was banging away on our common, non-square-root Friden. I heard a strange sound that went something like "punch-punch-cachunk, punch-punch-cachunk, punch-punch-cachunk-DING, punch-punch-DING-clang-clang" in a repeated rhythm. I thought my mate had either lost his marbles, or was creating some new kind of computer game.

I asked him what the heck he was doing. He said, "Finding a square root."

"But, um...," I said, "this isn't a square root Friden."

"I know," he replied. "That's why I have to do it this way."

Turns out, my officemate, "Gap," was an old NASA hand, and one who could make the Friden do all kinds of tricks, including the famous "Friden March," which was the division of two magic numbers that made the thing simulate the "rah, rah, rah-rah-rah" sound of a football cheer.

Somewhere along the line, Gap had learned the Friden algorithm for finding a square root on a non-square-root Friden. It was the same algorithm, in fact, as the one programmed into the cam-and-gear "ROM" of the square-root Friden. Gap taught it to me, and now I'll teach it to you. (Gap, if you're still out there and reading this, thanks for the education.)

The whole algorithm is based loosely on Equation 2, but written:

$$(n+1)^2 = n^2 + n + (n+1) \quad (3)$$

The keys on the Friden keyboard were "sticky," meaning that they held whatever was punched into them. The nice part about the formulation in Equation 3 is that the last digit punched in was also the number whose square you were taking. To see it work, watch the squares develop in Table 2.

Table 2 -- The "Friden" sequence.

Initial Number	Numbers Added	Result
0	0+1	1
1	1+2	4
4	2+3	9
9	3+4	16
16	4+5	25
25	5+6	36
36	6+7	49
49	7+8	64
64	8+9	81

As you can see, the number in the third column of this table is always the square of the last number added in.

Only Gap wasn't really adding, he was subtracting--from the number whose root he was seeking. The "DING" sound I had heard occurred each time the subtraction caused an overflow. This was the signal to Gap that he'd gone too far, so he'd back up by adding the last two numbers back in the *reverse* order he'd subtracted. In that way, the number remaining in the keyboard was always the largest digit whose square didn't exceed the input number. This is precisely the number we seek for the integer square root.

Shifting

This next part of the algorithm is critically important, both for the Friden solution and for our computerized ones. It's the notion of shifting, in a manner similar to that used in long division.

Somewhere in the dim recesses of your memory, you have a vague recollection of "pointing off by two" to begin the computation of a square root by hand. That's because squaring a one-digit number (from zero through nine) produces a two-digit result (zero through 81). Consider, for a moment, a two-digit decimal number, where the two digits are p and q. We can write this number as:

$x = 10p + q$ (4)

Squaring gives:

$x^2 = (10p + q)^2 = 100p^2 + 20pq + q^2$ (5)

To make the example more concrete, let:

$p = 9; q = 5$ (6)

so $x = 95$ (7)
and $x^2 = 100*81 + 20*45 + 25$
$= 8,100 + 900 + 25$
$= 9,025$ (8)

From this example, it's easy to convince oneself that the p^2 and the q^2 terms don't interact; the latter term can never be as large as 100, so it can never affect the value of the higher two digits--it fits neatly into the space reserved by the trailing two zeros of $100p^2$.

The middle term, 900 in our example, is a little more worrisome. It clearly overlaps into the high half of the result, and changes the values of the two high digits. However, a little reflection will convince you that the value of q can never influence the result we get for p. The worst-case condition occurs when $q = 9$. Even for this case, however, the sum can never be as large as the square of the next larger p. In mathematical terms,

$$10p + 9 < 10(p + 1) \quad (9)$$

Clearly, there is nothing magic about our choice of ten as the base for this number, so extension to other bases is straightforward.

The implication of Equation 9 is profound: it means that we can solve for each individual digit of the root separately; there is no need to backtrack and adjust those digits previously found. As we shift to the next lower digit (using the next pair of digits of the input number, which is why we point off by twos), we are refining the final result by adding digits to the right-hand side of the result, but those digits already found remain inviolate. This characteristic is what makes the whole Friden algorithm fly. To see it work, watch the square root develop (less the successive subtraction) in Table 3.

Table 3 -- Decimal square root.

```
            12        34                56                    78

          -0-11
             1
           -1-2
             8
           -2-3
             3     334
                        -30-31
                   273
                        -31-32
                   210
                        -32-33
                   145
                        -33-34
                    78
                        -34-35
                             9          956
                                      -350-351
                                           255          25578
                                                     -3510-3511
                                                          18557
                                                     -3511-3512
                                                          11534
                                                     -3512-3513
                                                               4509
```

As you can see from the table, we're still subtracting numbers in pairs, each one larger than the other. Note, however, that we don't ignore the higher-order digits, as we would in long division. Each of our subtracted numbers includes all the digits previously found, and we're working our way up through the last digit, starting with the zero. This process turned out to be just the right approach for the Friden calculator, which had sticky digits and thus remembered the leftmost digits already found. The nicest part of the whole approach, from the point of view of utility, was that after the last digit was found, the numbers left stuck in the keyboard did, in fact, represent the square root--in this case:

$$\sqrt{1234567890} = 35136 + fractionalpart \quad (10)$$

It's worth mentioning that this may be the only case in computing history where the result of the computation is found entered into the keyboard, instead of the usual result display.

Another subtlety of the Friden algorithm is also worth mentioning: in Table 3, we shifted the "dividend" by two digits to tackle the next digit of the result. This meant copying the digits found thus far over one digit to the right. In the real Friden algorithm, we don't want to move the digits already found--they're stuck in the keyboard. To avoid moving them, we shift the carriage one place to the left instead of two, which leaves the digits of the root lined up correctly without having to move them. You'll see a similar behavior in our final, binary version.

Finally, note that, unlike the case of division, the number we're subtracting gets more and more digits in it as we go, growing from one digit to five. This behavior is hardly surprising. In general the number of digits in the result, which is also the thing we're subtracting, is half that of the original argument. If we're to get this multi-digit result in the keyboard, it stands to reason that the number of digits involved in the subtractions must grow as the process unfolds. Describing this process in words, however, can't begin to evoke the sound that the Friden calculator would generate while taking a square-root.

From a practical point of view, however, the result is the sound of a thrashing calculator whose decibels, not to mention the complexity of the sound, are rising at an alarming rate, roughly comparable to the sound of a Yugo being seriously overrevved. Imagine the sound of a pondful of hoarse frogs, as one after the other chimes in to harmonize with the rest. The first time you hear it, it seems a toss-up which will arrive first, the solution or the demise of a calculating engine. However, despite the alarming noise, I never saw a Friden fail or give an incorrect result.

The High-School Algorithm

As I mentioned earlier, you and I both know that a method exists for the pencil-and-paper computation of the square root--we all learned the method, and promptly forgot it, in high school.

As fascinating as the Friden algorithm might be, and as valuable from a historical point of view, it has little if any value using any other calculator, one that lacks the sticky, matrix-like keyboard of the Friden. Nevertheless, it is the basis for the "high school" algorithm and gives us enough insight into the process so that, with any luck, you will be able to remember it this time.

Let's go back to Equations 4 and 5, which I'll repeat here for convenience:

$x = 10p + q$ (4)

$x^2 = 100p^2 + 20pq + q^2$ (5)

We'll assume that we've already managed to find p, by some method or another. We're now looking for the next digit, q. Rewriting Equation 5 gives:

$20pq + q^2 = x^2 - 100p^2$
$q(20p + q) = x^2 - 100p^2$

or

$$q = \frac{x^2 - 100p^2}{20p + q} \quad (11)$$

This formula gives us a rule for finding q. Notice that the numerator of the right-hand side is the original number, less the square of p, shifted left so that the two line up. In short, the numerator is the *remainder* after we've subtracted our initial guess for the root. To get the denominator, we must double our guess p, shift it left one place, and add q.

At this point, if you've been paying attention, you're asking, "How can I add q until I know what it is?" The answer, of course, is that you can't. This makes the square root algorithm a bit of a trial-and-error process, just like the division algorithm (hardly a surprise, there). In division, as we've learned so laboriously over the last few months, we look at the first couple of digits of the dividend and divisor, and use them to guess at the next digit of the quotient. We can't be sure that it's correct, however, until we've multiplied it by the entire divisor, and verified that we get a small, positive remainder.

In the same way, in the square root process, we assume a value for q, based on a divisor of $20p$. Then we must substitute the new value of q, and make sure the division still works.

It seems complicated, but it's actually no more complicated than division, and in some

ways it's a bit easier.

When I first wrote down Equation 4, I was assuming that p and q were single digits. However, you'll note that there's nothing in the math that requires that. We can now let p be all the digits of the root that we've found so far, and q the one we're seeking next. In division, the trial quotient digit must be tested at each step, and often adjusted. The same is true in the square root algorithm. However, in the latter, the larger p is, the less influence q will have on the square, and the less likely we are to have to backtrack and reduce q. Therefore, although we must still always check to be sure, there's actually less backtracking in the square root. The only thing that makes the algorithm seem harder is the fact that the root, and therefore the differences, are getting larger as we go along, just as we saw in Table 3.

Enough talk; let's see an example. We'll use the same input value we used before. Our first step is to point it off by twos:

12 34 56 78 90

Next, we write down the first digit of the root. We don't need a fancy algorithm for this. The first piece of the argument is only two digits long, and its root can only be one of nine numbers. Even a mathophobe should be able to figure out, by inspection, which of the nine it is. In this case, we see that four gives a square of 16, which is too large. So the first digit must be three. Write it down above the input value. Now square it, and subtract it to get the remainder:

```
  3
 ---------------
12 34 56 78 90
 9
 -
 3 34
```

As you can see, we've "brought down" the next digits, just as we do in division. The only difference is, we bring them down by twos, so our next dividend is 334. This is the top half of the division in Equation 11.

Now, here comes the tricky part: Remember that the bottom half is not $10p+q$, but $20p+q$. Before we look for q, we must *double* the current root, then tack on a new zero (bet that's the part you forgot!). At this point, the process looks like this:

```
    3
   ---------------
12 34 56 78 90
 9
 -
60| 3 34
```

Our division is, then, 334/60, which yields five and some change. Before we write down the new digit, five, however, we must make sure that the division still works when we

stick the five into the divisor. So we now have:

335/65,

which still gives us a five. We're okay, and don't need to backtrack. Write this next digit down, so our trial root is now 35. Please note: it's very, very important for you to see that the trial root and the thing we divide with are not the same, because of that factor of two. The last digit is the same, but the rest of the divisor is double that of the root. At this point, our process looks like this:

```
     3  5
   ----------
12 | 12 34 56 78 90
   |  9
65 | --
   |  3 34
```

Now, you're probably thinking, what next? How do I subtract to get the new remainder?

We can't just square 35 and subtract it, because we've already subtracted the square of three.

Again, Equation 5 provides our answer. We have:

$$x^2 = 100p^2 + 20pq + q^2$$

which we have already written in the form:

$$q(20p + 1) = x^2 - 100p^2$$

We've also already performed the subtraction on the right--that's how we got the remainder to divide with, to find q. Now that we've found it, we must complete the subtraction by subtracting out the left-hand side. That is, the remainder we now must obtain is:

$$rem = x^2 - 100p^2 - q(20p + q) \quad (12)$$

In this example, q is five, and $20p+q$ is 65, so we subtract 325. Note that this step is identical to division. After the subtraction, our process looks like this:

```
     3  5
   ----------
12 | 12 34 56 78 90
   |  9
65 | --
   |  3 34
   |  3 25
   |  ------
   |     9 56
```

At this point, we can begin to see how the algorithm works. Let's see if we can state it in

words:

- Point off the argument by twos
- Write down the first square root digit by inspection
- Subtract its square from the first two digits
- Obtain the remainder by drawing down the next two digits
- Double the square root, and append a zero
- Estimate the next root digit by dividing the remainder by this number
- Verify the digit by substituting it as the last digit of the divisor
- Multiply the last digit by the divisor
- Subtract to get the new remainder
- Repeat step five until done

By now, you should see how the process goes to completion. The completed process is shown in Figure 1.

Note again that when we subtract the next product, the last digit of the divisor must *always* be the same as the multiplier. That's because they're both q.

Is the result correct? Sure enough, if we square 35,136, we get 1,234,538,496. If we square the next larger number, 35,137, we get 1,234,608,969, which is too large. Thus the root we obtained, 35,136, is indeed the largest integer whose square is less than the original argument.

You've just seen the high-school algorithm in all its glory. Now that you've seen it again, you probably recognize at least bits and pieces of it. If, like me, you've tried to apply it since high school and found that you couldn't, it's probably because you forgot that doubling bit. The thing that you're dividing to get the next digit (the divisor) is *not* the root itself, nor is it exactly twenty times the root, either. You get it by doubling the root, then appending a zero. Also, before finally multiplying this divisor by the new digit, don't forget to stick that new digit into the divisor, in place of the original zero. When you do so, there's always the chance that the new product is too large. In that case, as in division, you must decrement the latest digit and try again.

Doing it in Binary

We now know (or, rather, we've been reminded) how to find square roots by hand, using decimal arithmetic. However, the algorithm is a lot more useful, and universal, than that. In deriving it, we made no assumptions as to the base of the arithmetic, except where we put a ten in Equation 4. Replace that ten with any other base, and the method still works. As in the case of division, it's particularly easy when we use base two, or binary arithmetic. That's because we don't have to really do any division to get the next digit. It's either a one or a zero, so we either subtract, or we don't. We know how to test for one number greater than another, and that's as smart as we need to be to make this method work in binary.

You'll recall that, in the decimal case, we made our first estimate of the quotient digit by using the divisor with a zero as the last digit. Then, once we had the new digit, we stuck it

into the divisor in place of the trailing zero, and tried again. Using binary arithmetic, we don't even have to do this step. If the division succeeds, it must succeed with the one as the last bit, so we might as well put it there in the first place.

In the example in Figure 2, we're going to repeat the long-form square root in binary. The input argument is:

45765 = 0xb2c5 = 0b1011001011000101

We perform exactly the same process as in decimal arithmetic. The only difference is that we don't have to do a division. We merely need to look at the magnitude of the two numbers, and either subtract, or don't.

The root is 0b11010101, which is 0xd5, or decimal 213. This result tells us that the square root of 45,765 is 213, which is true within our definition of the integer root. The next larger root would be 214, whose square is 45,796, which is too large. Any doubt we might have about the method is expelled by looking at the remainder, which is:

0b110001100 = 0x18c = 396,

which is also 45,765 - (213)2.

Implementing It

Now that you see how the algorithm works, let's try to implement it in software. The major difference between a hand computation and a computerized one is it's better to shift the argument so that we are always subtracting on word boundaries. To do this, I'll shift the input argument into a second word, two bits at a time. This second word will hold our partial remainders. We'll build up the root in a third word, and the divisor in a fourth. Remember, the divisor has an extra bit tacked on (the multiplication by two, remember?), so we need at least one more bit than the final root. The easiest thing to do is to make the divisor and the remainder words both the same length as the input argument. Listing 3 shows the result. Try it with any argument from zero through the maximum length of an unsigned long word (4,294,967,295), and you'll find it always gives the correct root. It gives it quickly, too--only 16 iterations, with nothing in the loop but a test and a subtraction.

As an aside, note that unlike in floating-point implementations, we don't test for the error of a negative argument. Because the argument is assumed unsigned, such a test is unnecessary.

Better Yet

The code of Listing 3 looks hard to beat, but it does use a lot of internal variables, doesn't it? It needs one to hold the remainder, into which we shift the input argument, one to hold the root we're building, and one to hold the divisor. Can we do better?

As a matter of fact, we can. The secret is in noting the intimate relationship between the root and the divisor. They differ only in that the root ends in one or zero, while the corresponding divisor has a zero tucked in front of the last bit, and so ends in 01 or 00. We can see this relationship by comparing the two, as I've shown in Table 4 for our example problem. The divisors for successful, as well as unsuccessful, subtractions are shown.

We can see that, whether the subtraction is done or not, the divisor is always four times the root, plus one. The success or failure of the subtraction only determines whether or not the bit inserted into the *next* root is a one or a zero.

Table 4 -- Root vs. Divisor.

Root	Divisor	Successful?
1	101	y
11	1101	n
110	11001	y
1101	110101	n
11010	1101001	y
110101	11010101	n
1101010	110101001	y

The nice part about this correspondence is this: because there's always a predictable relationship between the root and the divisor, we don't need to store both values. We can always compute the final result from the last divisor. The binary algorithm, modified to use a single variable for the divisor, is shown in Listing 4. In this implementation, we basically carry the desired information in the divisor. One division by two, at the end, gives us the desired root.

LISTING 3
The binary square root.

```
unsigned short sqrt(unsigned long a){
  unsigned long rem = 0;
  unsigned long root = 0;
  unsigned long divisor = 0;
  for(int i=0; i<16; i++){
    root <<= 1;
    rem = ((rem << 2) + (a >> 30));
    a <<= 2;
    divisor = (root<<1) + 1;
    if(divisor <= rem){
      rem -= divisor;
      root++;
    }
  }
  return (unsigned short)(root);
}
```

LISTING 4
An improved version.

```
unsigned short sqrt(unsigned long a){
  unsigned long rem = 0;
  unsigned long root = 0;
  for(int i=0; i<16; i++){
    root <<= 1;
    rem = ((rem << 2) + (a >> 30));
    a <<= 2;
    root ++;
    if(root <= rem){
      rem -= root;
      root++;
    }
    else
      root<;
  }
  return (unsigned short)(root >> 1);
}
```

Wrapping Up

Well, at this point we've pretty well said all there is to be said for integer arithmetic. In the past few months, we've covered multi-word addition and subtraction; we've looked at signed and unsigned multi-word multiplication; and we even found something I hadn't thought possible: a fast multi-word integer division. The square root algorithm you've seen here sort of puts the icing on the cake of integer arithmetic.

Since it's difficult to think of anything else we can do with integers, I'm sure you'd agree that we're well and truly done with them, and are ready to move on to other things.

You'd be wrong. Not too long ago, I ran across a remarkable algorithm for, of all things, the integer logarithm. It's just the ticket for cases where you have to simulate logarithmic behavior, for a loudness or brightness control, for example. I'll tell you all about this algorithm next month.

During these months, while we've been hammering away at integers, I've amassed a ton of other material to cover in future months. Lately I've been spending a lot of time fitting curves with nonlinear functions. I've also been looking at the generation of certain functions for simulation transients from input sensors and similar sources. We'll be talking about those in future months. I've also got the long-promised discussion of high-speed interpreters, and how they can be used in real-time systems.

In short, we have a lot of ground to cover now that integers are nearly behind us. I'm looking forward to it; how about you?

Jack W. Crenshaw is a staff scientist at Invivo Research in Orlando, FL. He did much early work in the space program and has developed numerous analysis and real-time programs. Crenshaw can be reached via e-mail at 72325.1327@compuserve.com.

Return to Embedded.com

Send comments to: Webmaster
All material on this site Copyright © 2000